Automated digital hardware synthesis using VHDL

Ailes, John W.

Monterey, California: U.S. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

AUTOMATIC DIGITAL HARDWARE SYNTHESIS
USING VHDL

by

John W. Ailes

September 1990

Thesis Advisor: C. H. Lee

Approved for public release; distribution is unlimited

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Department of Electrical and Computer Engineering 6b. OFFICE SYMBOL (*If applicable*) EC	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000	7b. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*)	10. SOURCE OF FUNDING NUMBERS Program Element No / Project No / Task No / Work Unit Accession Number

11. TITLE (*Include Security Classification*)
AUTOMATIC DIGITAL SYNTHESIS USING VHDL

12. PERSONAL AUTHOR(S) AILES, JOHN, W.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*year, month, day*)	15. PAGE COUNT
Master's Thesis	From To	September 1991	74

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18. SUBJECT TERMS (*continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUBGROUP	VHDL, DIGITAL, SYNTHESIS

19. ABSTRACT (*continue on reverse if necessary and identify by block number*)

The automatic synthesis of a hardware description language (HDL) representation of a digital device has been the subject of significant research in the past five years. This thesis explores this topic as it applies to finite state machines and combinational logic expressed in a subset of the IEEE standard language VHDL (VHSIC Hardware Description Language). It describes the subset chosen, and the development of VHDL2PDS, a program which automates the process of translating VHDL to PALASM, a hardware synthesis language. The PALASM description is then directly implemented into a field programmable gate array (FPGA) using the Xilinx Logic Cell Array (LCA) development system. Complete examples are provided which illustrate top-down design and testing using VHDL, and the use of software to produce a FPGA. This thesis demonstrates that selected constructs in VHDL can be automatically synthesized with a resulting savings in engineering development time due to the simplicity of this approach and the ease of verifying the correctness of the design.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS REPORT ☐ DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL C. H. LEE	22b. TELEPHONE (*Include Area code*) 408-646-2056	22c. OFFICE SYMBOL EC/Le

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted SECURITY CLASSIFICATION OF THIS PAGE
All other editions are obsolete

Approved for public release; distribution is unlimited.

Automatic Digital Hardware Synthesis
Using VHDL

by

John W. Ailes
Lieutenant, United States Navy
B.S., Oregon State University

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 1991

Author: [redacted]
John W. Ailes

Approved by: [redacted]
C. H. Lee, Thesis Advisor

[redacted]
Chyan Yang, Second Reader

[redacted]
Michael A. Morgan, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

The automatic synthesis of a hardware description language (HDL) representation of a digital device has been the subject of significant research in the past five years. This thesis explores this topic as it applies to finite state machines and combinational logic expressed in a subset of the IEEE standard language VHDL (VHSIC Hardware Description Language). It describes the subset chosen, and the development of VHDL2PDS, a program which automates the process of translating VHDL to PALASM, a hardware synthesis language. The PALASM description is then directly implemented into a field programmable gate array (FPGA) using the Xilinx Logic Cell Array (LCA) development system. Complete examples are provided which illustrate top-down design and testing using VHDL, and the use of software to produce a FPGA. This thesis demonstrates that selected constructs in VHDL can be automatically synthesized with a resulting savings in engineering development time due to the simplicity of this approach and the ease of verifying the correctness of the design.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. OVERVIEW

This thesis examines the process of digital hardware design from the perspective of *synthesis*. Chapter I defines synthesis and introduces the concept of top-down hardware design using a hardware description language (HDL). The Institute of Electrical and Electronic Engineers (IEEE) standard language: VHSIC Hardware Description Language (VHDL) is covered in Chapter II, and it is contrasted with PALASM, a digital hardware synthesis language. Chapter III addresses the creation of a subset of VHDL that may be translated to the dialect of PALASM that is supported by the Xilinx Logic Cell Array (LCA) development system. This allows the engineer to use VHDL to create and validate a design, and then to implement it in a gate array. The development of software to translate VHDL combinational logic and finite state machine descriptions to PALASM follows in Chapter IV. Chapter V examines the structure of the Xilinx LCA and the associated software used to customize this gate array. Next, a design paradigm using VHDL to develop, simulate, and synthesize digital hardware is presented in Chapter VI. Finally, conclusions and areas for future research are considered in Chapter VII.

B. DIGITAL SYNTHESIS

Digital synthesis is the process of creating a digital system which exhibits a desired behavior. However, the process of digital synthesis is more than a single step. It consists of a range of tasks, beginning with a very general definition of the desired behavior of the device and concluding with the physical layout of the transistors used to implement it. To classify these tasks, we can create a hierarchy of levels associated with a digital system. Figure 1 shows a proposed hierarchy. Synthesis is the process of moving from higher levels to lower levels in the hierarchy.

1. The Behavioral Level

The algorithmic or behavioral level is the highest tier in our hierarchy and therefore the most abstract. At this level, the functioning of the device is expressed in terms of the *desired* behavior. No detail is given to *how* the behavior will be achieved.

2. The Architectural Level

The next layer describes the structure that will be used to accomplish the desired behavior. It may specify whether a register, adder, shifter, or other component will be used and how these large blocks will interact. However, it is more detailed then the algorithm described in the behavioral level.

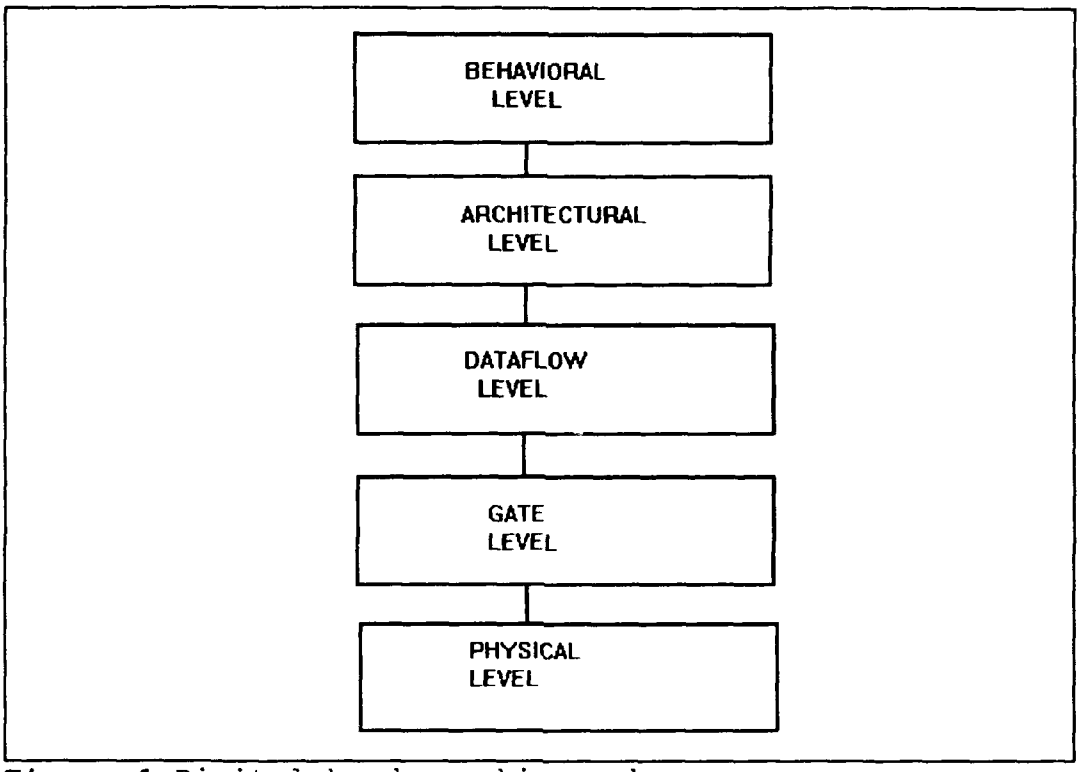

Figure 1 Digital hardware hierarchy

3. The Dataflow Level

The dataflow level implements the algorithmic behavior on the architecture. It also known as the *register transfer level* and it specifies precisely how data will flow among the components in the design architecture.

4. The Gate Level

The gate level describes the logic gates that implement the architecture and behavior. It is most often illustrated in the form of a schematic, but it may be represented in another computer-readable form. In the later case, computer-aided design (CAD) tools are available which can store and manipulate the gate level descriptions.

5. The Physical Level

The lowest level in the digital description hierarchy is the physical level. This level describes the transistor layout used to implement the design. At the physical level, one may have a VLSI layer representation such as that available from the Magic tool set [Ref. 1:p. 109], or it may be a gate array configuration stored in Erasable Programmable Read Only Memory (EPROM) or a fusible link map used with a programmable array logic (PAL) device.

C. BOTTOM-UP DESIGN

Traditional digital design begins by considering the fundamental building blocks available. In some cases, this may be at the gate level using small-scale integration (SSI) circuits. More often, medium-scale integration (MSI) chips are selected from a manufacturer's data book. Alternatively, the designer may use a computer-aided design (CAD) system that allows him to select devices from a library of standard parts. In all these cases, the problem is approached from a *bottom-up* perspective; the system is constructed through the interconnection of a fabric of extant parts. The principal advantage to the approach is that it allows the designer to simplify his task through the use of proven components. Moreover, it is a method that is widely accepted and familiar to many digital designers. However, it requires the designer

to have insight into the organization of the final design which can be built from the component parts available to him.

D. TOP-DOWN DESIGN

In contrast to the bottom-up technique of digital design, top-down design does not consider the components available as a first step. Instead, the approach is to, "[Start] with the specification of the complete system in a form compact enough that one person can readily comprehend it." [Ref. 2:p.3] This *high level design* is then divided into smaller tasks that together make up the original design objective. In turn, each of these smaller subdesigns can themselves be partitioned into devices that are still smaller in scope. When fully partitioned, the subdesigns should be straight-forward to implement. However, a problem arises in the synthesis of the small designs into a larger device; designs that are able to function correctly by themselves may fail when interfaced together. This problem may be alleviated if the design has very precisely specified interface descriptions. Once this obstacle is surmounted, there are many benefits to top-down design. The biggest advantage is that it promotes *abstraction*; the designer can think more about the design of the system and less about the bare hardware used to implement it. A second major advantage is that the subsystems are more logically partitioned. This means that they are more easily understood and are therefore easier to debug.

E. HARDWARE DESCRIPTION LANGUAGES

Hardware description languages are designed to simulate digital hardware. The hardware modeled may be an extant device, or one may test a design that has not yet been built. The ability to model devices without creating a prototype greatly streamlines the task. It allows the engineer to create a high level description of a device and to interactively study the effect that changes in the high level design have on its behavior and performance. The result is a tremendous savings in time and a greatly improved design. Moreover, this top-down approach can be applied to all facets of the design process. By introducing *simulation* early in the design cycle, changes may be made to components without costly redesign of the subsystems. As the design reaches completion, greater detail is added and the model more closely mirrors the behavior that the physical device will exhibit. Moreover, by rigorously specifying the interface between the subcomponents of a design, a task may be divided across work groups. Each group may create HDL descriptions of its piece of the larger design that can interact with the other component parts. Furthermore, once the design is complete, much of the work of introducing the device in a new technology will not be duplicated; the models can be readily changed and reused. In addition, designers may exchange design data in the form of HDL models and vendors may provide behavioral descriptions of their products that may be incorporated into one's design.

Finally, a hardware description language can provide a greater range of testing than even the most robust digital integrated circuit tester. Since the test software is written in the same language as the model itself, it may interact with the model and provide more thorough testing than the single input stimulus files that are used to test digital devices.

II. AN OVERVIEW OF VHDL AND PALASM

A. VHDL

VHDL is rapidly emerging as *the* standard hardware description language. It is both a Department of Defense (DOD) and Institute of Electrical and Electronic Engineers (IEEE) standard. It is a very rich language, encompassing most of the features normally associated with a high level language. In addition, it has many features specifically designed to facilitate digital hardware simulation across the synthesis hierarchy, from the behavioral to the gate level.

1. The Standard Hardware Description Language

The IEEE sponsored some of the earliest meetings on the subject of hardware description languages. The first of these was held at Rutgers University in 1973. It was followed by full conferences in New York in 1975 and Palo Alto, California in 1979. [Ref. 3:p. 1] The Department of Defense became interested in the topic of hardware description languages to facilitate work being done in its Very High Speed Integrated Circuit (VHSIC) program. In June, 1981, DOD held its first meeting to create a new standard HDL. In July 1983, DOD awarded a contract to develop VHDL to a team consisting of IBM, Texas Instruments, and Intermetrics. [Ref. 4:p.94] Working in close cooperation with the IEEE, VHDL was developed

as the Department of Defense's standard hardware description language. In March, 1987, the IEEE also adopted VHDL as its standard HDL in IEEE-1076. [Ref. 5:p. 2]

2. Features of VHDL

VHDL was designed to be very similar to the DOD standard general purpose programming language **Ada** and it supports many of the same constructs. In addition, it has many supplemental features designed to facilitate digital hardware design.

a. Classes of Objects

VHDL has three classes of objects: signals, variables, and constants. A signal corresponds to the wires in a hardware design. It may be assigned a value, but that value does not take effect immediately. Instead, it is *scheduled* by the VHDL simulator to take on the value at a future time in the simulation cycle. Variables are used by the model to facilitate simulation, but they have no analogue in the physical hardware. Unlike signals, there is no propagation delay in a variable assignment; it takes place instantaneously. Finally, constants are assigned their values when the program is compiled and never change during its execution.

b. Strong Typing

VHDL follows the convention of *strong typing*. *This* means that each signal, variable, and constant in the program

must be declared to be of a certain type and objects of different types may not be freely intermixed. The type of an object tells the compiler what kind of data will be stored in it. Examples of types include integer, floating point, character, and boolean. In addition, the user may introduce his own user-defined types. A *composite type* is an object that contains other types in its definition. For example, a **record** is a type that contains **fields** that are themselves composed of other types.

c. Dynamic Data Structures

Like Ada, VHDL supports *dynamic data structures* that allow a VHDL program to allocate additional space for variables *while the program is executing*. In languages such as FORTRAN which support only static objects, space may only be set aside for variables at the time the program is compiled. VHDL supports a construct known as an **access type** which is a pointer to a data structure. As the VHDL program is running, space may be allocated for a variable through the use of the reserved word **new**.

d. Subprograms

VHDL supports *subprograms* that allow partitioning of a program. There are two types of subprograms in VHDL: procedures and functions. The difference between procedures and functions is that a function *returns a value* and is therefore part of an expression. Moreover, functions are

designed so as not to alter the value of objects; instead they return a value that is dependent on other objects. In contrast, a procedure does not return a value and therefore may not be used in an expression. However, it may change the value of an object if the object is declared to be of mode **out** or **inout**. Procedures may not alter the value of an object declared to be of mode **in**. All parameters passed to a function are implicitly defined to be of mode **in**. Figure 2 contains the subprogram body of an example function and a procedure.

e. Packages

The idea of a **package** is borrowed from Ada. It allows a collection of functions, procedures, and declarations to be grouped together in a single entity. There are two parts to a package: the declaration section and the body. The declaration section specifies the portion of the package that is visible outside the package. The body contains declarations that are not available outside the package and it also contains the detailed definition of the algorithms that implement the functions and procedures that are visible outside the package. Figure 3 contains an example of a package.

f. Libraries

Libraries in VHDL contain the designs that the user or others have created. A single design may span several

```
function Bigger(A, B:Integer) return Boolean is
-- This function returns TRUE if A>B
begin
  if A>B then
    return TRUE;
   else
    return FALSE;
   end if;
end Bigger;

procedure swap(A,B: inout Integer) is
-- This procedure interchanges A and B
variable temporary:integer;
begin
  temporary:=A;
  A:=B;
  B:=temporary;
end swap;
```

Figure 2 Example VHDL function and procedure

libraries. In addition, there are two standard libraries: **WORK** and **STD**. **WORK** holds the current design and **STD** contains the system-provided packages **STANDARD** and **TEXTIO**. **STANDARD** defines the standard types available in VHDL and it allows the user access to the current simulation time. **TEXTIO** contains the routines and declarations needed to perform console and file input and output. VHDL provides the keyword **USE** to allow the designer access to libraries outside of the current design.

```
package Example is
  type example_type is ('a','b','c','d');
  function Max(a,b:Integer) return Boolean;
end Example;

package body Example is

  function Bigger(A, B:Integer) return Boolean is
  -- This function returns TRUE if A>B
  begin
    if A>B then
      return TRUE;
    else
      return FALSE;
     end if;
   end Bigger;

end Example;
```

Figure 3 Example VHDL package

3. Differences between VHDL and other High Level Languages

In some aspects, VHDL is very similar to other high level languages. None of the features mentioned so far are markedly different from those found in Ada, Pascal, or C. However, there are some profound differences.

a. VHDL as a Modeling Language

"The general model on which VHDL is based consists of three interdependent models: a behavioral model, a timing model, and a structural model." [Ref. 5:p.7] These three components are the heart of the modeling capability of VHDL.

(1) The Behavioral Model

Figure 4 provides an example of a VHDL behavioral model. This model allows the designer to describe the functionality of a device without the constraint of providing the timing that corresponds to it. The behavior is algorithmic in nature. The output is obtained by the simulator executing the program steps. This process need not have any resemblance to the way the hardware actually will perform the desired action; it simply mimics its behavior.

(2) The Timing Model

An example of the timing or *dataflow* model is provided in Figure 5. It differs from the behavioral model in two ways. First, as Steve Carlson of Synopsys, Inc. notes, the dataflow level has, "an implied architecture." [Ref.6:p.3] The equations which implement the design in the dataflow model can be realized in hardware directly. This is very different from the behavioral model which only *emulates* the behavior of the hardware. The second difference is that timing information may be included to more precisely model the design's behavior. If no timing parameters are supplied, a *delta* delay is assumed. Delta delays are discussed below.

(3) The Structural Model

While the behavioral and dataflow models both deal with the functioning of the design, the structural model describes how the design can be constructed from other

```
architecture behavioral_description of jkflip_flop is
   SIGNAL current_q : t_wlogic := fX;
begin
 p1:process
 begin
   if j = f0 and k = f0 then
     q <= current_q;
   elsif j = f0 and k = f1 then
     q <= f0;
   elsif j = f1 and k = f0 then
     q <= f1 ;
   elsif j = f1 and k = f1 then
     q <= not current_q;
   end if;
   qbar <= not current_q;
   wait on clock;
end process p1;

p2:process(reset)
  begin
    if reset = f0 then
       q <= f0;
      qbar <= f1;
    end if;
   end process p2;
end behavioral_description;
```

Figure 4 Example VHDL behavioral description

devices. The **entity** construct contains the interconnection data in a **port** statement that specifies the direction of dataflow, and its type. For example, it may specify that a certain signal is a bit, or alternatively that it is a complex number. The amount of abstraction is at the discretion of the designer. Figure 6 shows an example of the structural component of a VHDL design.

```
architecture dataflow_description of adder is
begin
  process
  begin

F(0) <= transport (not CN)  xor  ((not (A(0) or (((not M)
and B(0)) or (M and (not B(0))))))  xor (not(A(0) and ((
not M) and B(0)) or  (M and (not B(0))))))  ) after 80NS;

F(1) <= transport (not ((not(A(0) or (((not M) and B(0))
or (M and (not B(0))))))) or ((not (A(0) and (((not M) and
B(0)) or
(M and (not B(0))))))) and CN))) xor ((not (A(1) or ((( not
M) and B(1)) or (M and (not B(1))))))) xor (not ( A(1) and
(((
not M) and B(1)) or (M and (not B(1)))))))) after 80NS;

  end process;

end dataflow_description;
```

Figure 5 Example VHDL timing model

4. The VHDL Simulation Cycle

With a basic knowledge of the features supported by VHDL, we can now discuss the way VHDL models time. The fact that digital hardware is inherently parallel in nature poses some difficulty for the HDL creator who wishes to model this parallelism on his strictly sequential computer. Moreover, the approach one takes in overcoming this problem must be general enough to guarantee the same output no matter the type of computer it is implemented on. The approach the designers of VHDL took was to define a two-stage model of time called the simulation cycle. [Ref.5:p.12] The first stage of the two-stage model updates the values of all signals that have been scheduled to receive a new value at the current time in

```
entity ADDER is
  port(M    : in Bit;
       A    : in Byte;
       B    : in Byte;
       CN   : in Bit;
       F    : out Byte);
end ADDER;
```

Figure 6 Example VHDL structural description

the *simulation* clock. The second stage involves the execution of *processes*. Processes are the tasks that the simulator schedules to occur at specific simulation times based on a *sensitivity list* of signals associated with each process that is either explicitly or implicitly specified by the user. When a signal that is on the sensitivity list of a given process changes, the process is scheduled to execute. The process continues to run until a **wait** statement is encountered. As discussed above, there is always a delay between the time a signal is assigned a value and the time it takes on that value. If the designer does not specify the delay, a *delta* delay is assumed. Assignments for which the user specifies a delay cause the simulator's system clock to be updated. Conversely, delta delays do not update the system clock.

5. The VHDL Test Bench

Another significant feature of VHDL is the ability to create a design entity that can interact with a hardware

design. This entity is known as a *test bench* and it provides a great deal more than a simple stimulus file and resulting output. This stems from the fact that the test bench can modify the stimuli it provides to the model based on the output of the model. For example, if the model being tested is an arithmetic logic unit, the test bench can iteratively verify the correctness through successive tests. If it discovers an error in the process, it can provide additional input to the model to identify the source of the problem. Moreover, the test bench may be run throughout the design cycle. If an error is encountered when a new revision is added, it can be flagged immediately, greatly cutting down the time required to debug the system.

B. PALASM

PALASM is a simple hardware synthesis language that was originally designed to be used with Monolithic Memories's programmable array logic (PAL). When Monolithic Memories was purchased by Advanced Micro Devices, the development of PALASM continued. However, Xilinx uses an older version of PALASM with their state of the art Logic Cell Array (Logic Cell is a Xilinx trademark) gate array system. The Xilinx version of PALASM supports only two constructs: combinational logic assignments and registered assignments. A design created in Xilinx's PALASM can use up to the full 20,000 gate maximum currently supported by the largest LCA. In addition, logic

minimization based on the University of California, Berkeley's Espresso program is provided [Ref.1 :p. 97]. Unlike VHDL, PALASM is not a hardware description language; it is not designed to model hardware. Instead, PALASM is designed to be synthesized into hardware. As seen in the PALASM state machine presented in Figure 7, the syntax is quite simple. It begins with the documentary information about the design including title, pattern, revision, author, company, and date. The next statement is the **CHIP** statement that declares the type of chip being used. In the Xilinx system, this is simply "LCA" for Logic Cell Array. After the **CHIP** statement, there is a pin declaration statement that declares the names of signals which will be assigned to pins on the gate array. The order is unimportant in the Xilinx system. Next there is the keyword **EQUATIONS** which indicates the start of the equation section of the program. Equations which have a simple equal sign are combinational logic expressions. Assignments with a colon followed by an equals sign are *registered* equations. They take on the value of the expression on the right side of the equation only on clock edges. Each must have an accompanying ".CLKF" statement to indicate which pin name provides the clock. Optionally, a reset line may be indicated with the use of ".RSTF" appended to the signal name.

```
;-----------------------------------------------------
TITLE        STATE_MACHINE.PDS
PATTERN      A
REVISION     REV 1.0
AUTHOR       Automated
COMPANY      U.S. Naval Postgraduate School
DATE         7/31/1991
CHIP STATE_MACHINE LCA
;---  PIN Declarations-----------------------------------
A B C D E F G H I CLK RESET ST0 ST1 ST2 ST3 ;
;-----------------------------------------------------
EQUATIONS
;----- Combinational Equations -------------------------
I=/CLK
;- State Equations ----------------------------------------
ST3:=(A*ST0*/ST1*/ST2*/ST3)+(/A*B*C*D*/ST0*ST1*ST2*ST3)+(
/B*C*D*/ST0*ST1*ST2*ST3)+(/C*D*/ST0*ST1*ST2*ST3)+(/D*/ST0
*ST1*ST2*ST3)+(A*B*C*/ST0*ST1*ST2*/ST3)+(/A*B*/ST0*ST1*/S
T2*ST3)+(/B*/ST0*ST1*/ST2*ST3)+(A*/B*C*/ST0*ST1*/ST2*/ST3
)+(D*/ST0*/ST1*ST2*/ST3)+(/C*/ST0*/ST1*/ST2*ST3)+(B*/ST0*
/ST1*/ST2*/ST3)

ST2:=(/A*B*C*D*/ST0*ST1*ST2*ST3)+(/B*C*D*/ST0*ST1*ST2*ST3
)+(/C*D*/ST0*ST1*ST2*ST3)+(/D*/ST0*ST1*ST2*ST3)+(/ST0*ST2
*/ST3)+(A*B*/ST0*ST1*/ST2*ST3)+(C*/ST0*/ST1*/ST2*ST3)

ST1:=(/A*B*C*D*/ST0*ST1*ST2*ST3)+(/B*C*D*/ST0*ST1*ST2*ST3
)+(/C*D*/ST0*ST1*ST2*ST3)+(/D*/ST0*ST1*ST2*ST3)+(/ST0*ST1
*/ST3)+(/ST0*ST1*/ST2*ST3)ST0:=(/A*ST0*/ST1*/ST2*/ST3)+(A
*B*C*D*/ST0*ST1*ST2*ST3)

E=(ST0*/ST1*/ST2*/ST3)
F=(/ST0*ST1)
G=(/ST0*ST2)
H=(/ST0*ST3)
ST0.RSTF = RESET
ST0.CLKF = CLK
ST1.RSTF = RESET
ST1.CLKF = CLK
ST2.RSTF = RESET
ST2.CLKF = CLK
ST3.RSTF = RESET
ST3.CLKF = CLK
```

Figure 7 Example PALASM state machine

III. SYNTHESIZING VHDL

This chapter presents an overview of our approach to the problem of synthesis. It discusses the tradeoffs made and the philosophy of our method of synthesis. Further details are provided in Chapter 4, which describes the software created based on this approach.

A. CREATING A SUBSET OF VHDL FOR SYNTHESIS

To begin with, VHDL was designed as a simulation language, not as a synthesis language. Therefore, some constructs in VHDL are not realizable in hardware. These include access types, records, user-defined types and recursive subprograms. This means that a subset of VHDL must be developed. Wolfgang Glunz of the Siemens company in Germany notes, "Most of the subsets [currently defined] are restricted to VHDL data flow descriptions of VHDL processes that can be synthesized with combinational logic." [Ref. 7:p. 1] These restrictions were followed in our synthesis effort. Our approach was to determine the subset based on our ability to translate it to PALASM. With this goal in mind, more than a subset specification was necessary; a corresponding *style* that defined how a description is laid out was required. Finally, our definitions were created with the expectation that they

will be built upon. The subset and style are created with room for more features to be added.

B. BEHAVIORAL VERSUS DATAFLOW

As noted earlier, the difference between behavioral and dataflow descriptions of a design is that the dataflow description has an implied architecture. Since our system is designed to be implemented on Xilinx LCA hardware, there is an inherent physical layout, but the architecture is in a very real sense user-configurable. To limit the scope of our design effort, we have chosen to implement dataflow descriptions of combinational logic and finite state machines. The description of a state machine nearly spans both the behavioral and dataflow domains. The description itself does not imply an architecture, but the registered equations which drive flip-flops in the Xilinx LCA hardware do have an explicit architecture. Behavioral descriptions are one level greater in abstraction and are therefore more general; they are not so closely tied to the hardware realization. Since all of our designs will be realized by translating the VHDL description to PALASM that is in turn synthesized by combinational gates and flip-flops, there is an implicit architecture. Therefore, all of our designs are expressed in VHDL as dataflow descriptions.

C. TRANSLATING COMBINATIONAL LOGIC

Purely combinational logic may be synthesized by direct translation from VHDL to PALASM. The style we have chosen in VHDL mimics the PALASM output file as closely as possible. The architectural information in the VHDL entity statement is used to provide the pin signal names in the PALASM code. Specifically, VHDL **port** statements are translated to PALASM pin assignments. The architectural dataflow description provides the equations that are then translated to PALASM syntax.

D. TRANSLATING STATE MACHINE DESCRIPTIONS

The task of translating a VHDL state machine description to PALASM is more difficult than that of combinational logic translation. A mechanism for generating the register transition and output equations is required. The approach followed was to use an intermediate form acceptable to a *state machine compiler*. The state machine compiler creates transition and output equations from the state machine description provided to it. The style of our state machine description in VHDL was chosen to correspond as closely as possible to the input syntax of the state machine compiler. The state machine input format handled transition descriptions with **case** statements, but it was easiest to model these with VHDL **if** statements. Complexity in the **if** statements was handled with recursive calls to our compound conjunctions

routine. Once the VHDL description was successfully translated into the format acceptable to the state machine compiler, software was written to take the output format it created and translate it into PALASM syntax. This process is illustrated in Figure 8. As in the combinational logic case, the VHDL **port** statements were translated to PALASM pin assignments. Additionally, our style format allowed the intermixing of state machine descriptions and combinational logic. As noted above, the specific details of our software implementation are provided in the next chapter.

E. TIMING CONSIDERATIONS

The creation of a VHDL subset and style required consideration of the problem of allowing user-supplied timing information. The timing information must be easy to change, or the model loses the important characteristic of technology independence. To achieve this in the combinational description, the **after** statement that specifies the gate delay is optional. The delay may be specified as a VHDL **constant** that may be read from a technology library to make it easier to change. To obtain the actual value for timing, the Xilinx routing tools must be used and the design then simulated using a hardware simulator. The results of this simulation may be incorporated into the VHDL description. This process is known as *back annotation*. For the state machine description, the system clock provides synchronization. Therefore the state

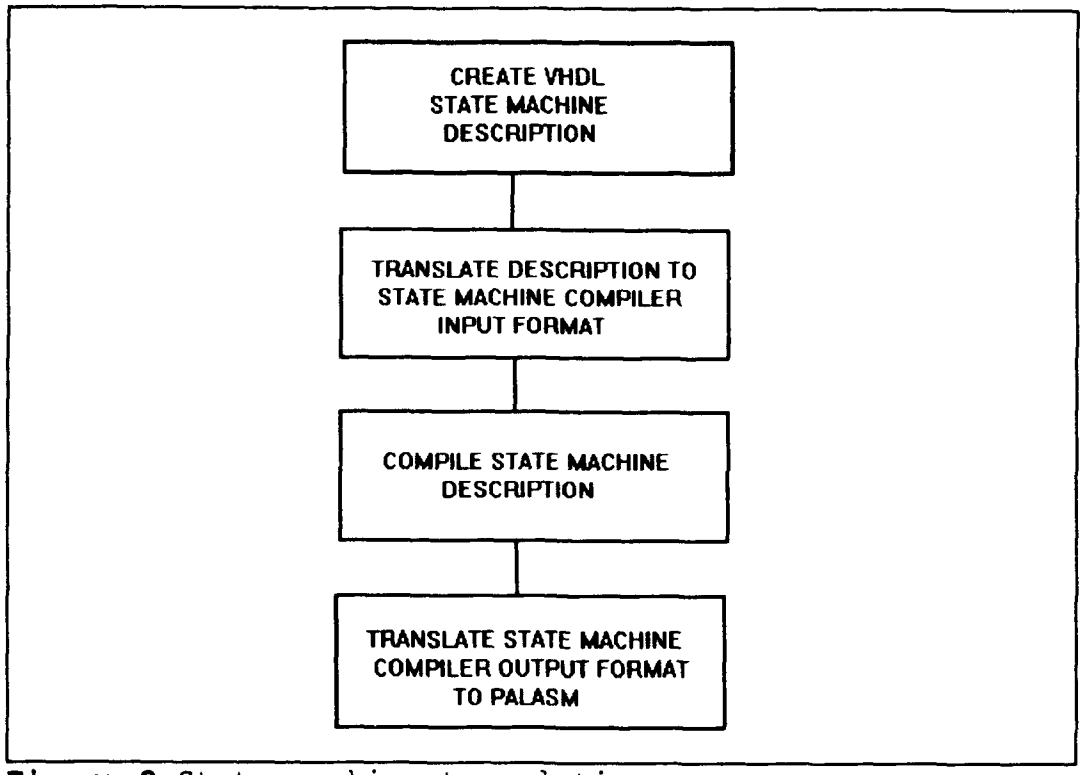

Figure 8 State machine translation process

machine could be created as a "metric-free timing representation" as discussed by Steve Carlson of Synopsys [Ref. 6:p. 68]. The idea is that since so much of the synthesized design is not known until the physical layout is created, only those facets of timing information that must be modeled are included. Hence, synchronous clock-driven designs like state machines may omit timing information in their dataflow descriptions.

IV. TRANSLATION OF VHDL DESCRIPTIONS TO PALASM

A. VHDL2PAL

VHDL2PAL is a stand-alone program that translates a VHDL combinational logic description into a corresponding PALASM representation. It is written in Borland Turbo Pascal, and it runs on an MS-DOS computer with at least 512K of RAM. Its major component routines are depicted in Figure 9.

1. VHDL Combinational Logic Syntax

The syntax of combinational logic expressed in VHDL accepted by VHDL2PAL is presented in Figure 10. The description must include a statement to use the library "work.translate.all" which contains the definitions for bit, byte, half word, word, and double word, all of which are based on a 32-bit word machine. These definitions are necessary for the VHDL simulator; the translator assumes their existence. The next portion of the description is the entity statement. The user should choose a descriptive name for the device he is modeling. VHDL2PAL will use this name in the synthesized PALASM code it creates. The next statement that appears is the port statement. The port statement lists each signal, indicates whether it is an input or output, and gives its size. Again, the synthesizer accepts sizes of bit, byte, half word, word, or double word. The dataflow description appears

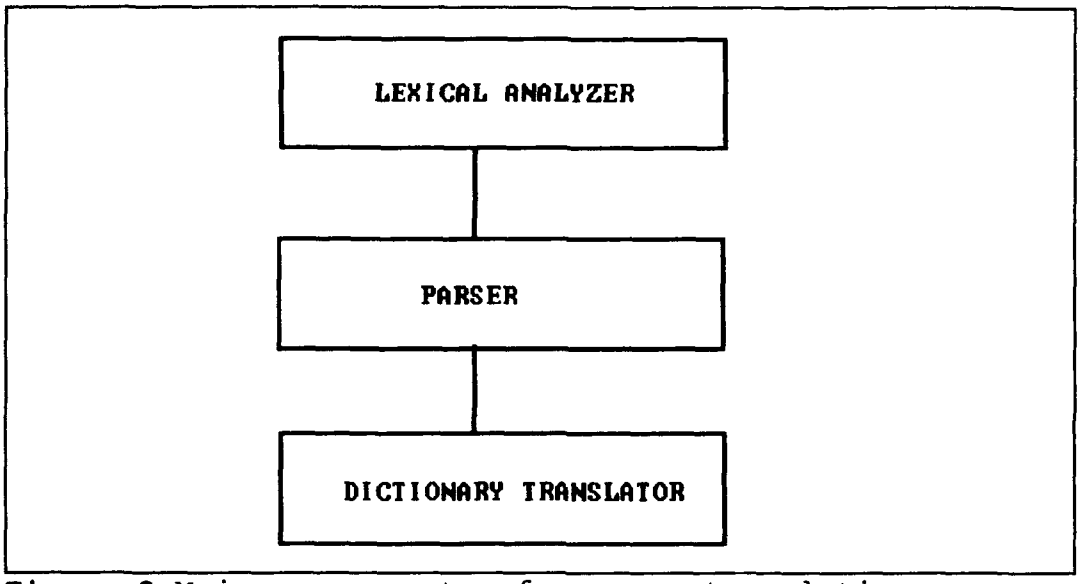

Figure 9 Major components of program translation

next. It contains a listing of each signal with its output. The keyword **after** followed by a numeric delay time may be placed at the end of a signal assignment. This permits the designer to model the length of time the hardware will require to perform the given function. The estimate he makes is based on his experience and is a function of the complexity of the expression. The actual delay will only be known after the design is synthesized and routed. Nevertheless, with experience, a reasonable estimate may be made.

2. The Translation Process

As noted earlier, the major components of the translation process are illustrated in Figure 9. Each are now covered in detail.

a. Lexical Analysis

The first phase in translating a VHDL description into PALASM is similar to that performed by a high level language compiler: the VHDL source code is separated into *tokens* [Ref. 8:p.6] which are groups of characters th- together make up a single atomic word. To perform this function, VHDL2PAL has a procedure named **next_token** that searches the input file and returns a string that, as the procedure's name suggests, is the next token in the input stream. The key to identifying a token is to know which characters separate the token from the characters around it. These characters are known as *delimiters* [Ref. 8:p. 77]. VHDL2PAL implements delimiters using the Pascal set construct. If a character is in the set **delimiter**, then procedure **next_token** determines that the current token is complete. The process of finding a token in an input stream is known as *lexical analysis* [Ref. 8:p.6].

b. Parsing

After the lexical analyzer find a token, it is examined to see if it is in the correct syntax. For example, the translator searches for the keyword "entity" when it is

determining the name the user has given his design. If the translator does not find "entity" where it expects to, a fatal syntax error occurs and the user is notified. The process of taking a token and deciding whether its presence does not violate the syntax of a language is known as *parsing* [Ref. 8:p. 146]. After VHD2PAL finds the entity name, it next parses the port statement. If the signals it reads in the port statement are of type byte or larger, the program expands its list of pins to allocate the requisite number of pins for the given data size. For example, if a signal is of type word, 32 pins are allocated in the PALASM design file.

c. Dictionary Translation

When VHDL2PAL has successfully read in all of the signals, it next parses the dataflow architecture description. It checks to see that the entity name is the same as previously declared. It then proceeds through the input file checking each output variable to see whether it is in the list of signals declared in the **port** statement. If the program encounters a variable that has not been declared, the program terminates with a fatal error. The program also examines each token it finds to see if it is present in the dictionary of tokens that can be directly translated from VHDL to PALASM. For example, if it finds the VHDL construct "XOR" it substitutes the symbol ":+:" which is its PALASM counterpart. The program continues parsing the input text, checking each

variable against those that have been declared, and directly translating symbols it finds in its dictionary. The program terminates when it encounters the VHDL program's final "end" statement.

B. VHDL2PDS

VHDL2PDS is a batch file that invokes three subprograms which handle the translation of a VHDL state machine description into PALASM. These three subprograms are VHDL2PEG, PEG, and EQN2PDS as shown in Figure 11.

1. VHDL State Machine Syntax

The syntax of a state machine description in VHDL acceptable to VHDL2PDS is presented in Figure 12. Although the entity definition is very similar to the combinational logic syntax previously discussed, the dataflow description is markedly different. Like the combinational syntax, the state machine description must include a **use** statement that directs the VHDL compiler to read the translation library. Also, any additional libraries that the user desires may be included. All of the signals to be used are again declared in the **port** statement. However, two special signals also must be declared: a reset signal, and a clock signal. Since a state machine must have a clock, and it must start in a known state, it is not surprising that these signals must be provided. The dataflow description begins with the assignment of combinational variables. Any number of variables may be

```
use work.translate.all;     -- Translation library
[use work.other.all;]       -- Optionally use others
entity <entity_name> is
   port (<variable1> : <type> <size>;
   <variable2> : <type> <size>;
                .
                .
                .
   <variableN> : <type> <size>);
 end <entity_name>;

 architecture Dataflow_Description
   of <entity_name> is
   begin
   process(<variable list>)
   begin
    <out_var1> <= transport <expr> [after<time>];
    <out_var2> <= transport <express> [after <time>];
          .
          .
          .
    end process;
end Dataflow_Description;
```

Figure 10 VHDL combinational logic syntax

placed here, up to the limit of the physical gate array. The next statement of great importance is the state type declaration. This tells the translator how many state variables to set aside. Next the state variable is declared. It is important to initialize this variable to zero, as the LCA device always assumes state zero on power up. After the state variable is declared, the output equations follow. The state machine must follow the Moore model; its outputs are only a function of the state of the machine. For each state, the output of each variable must be specified. After all of the output variables have been specified for every state in

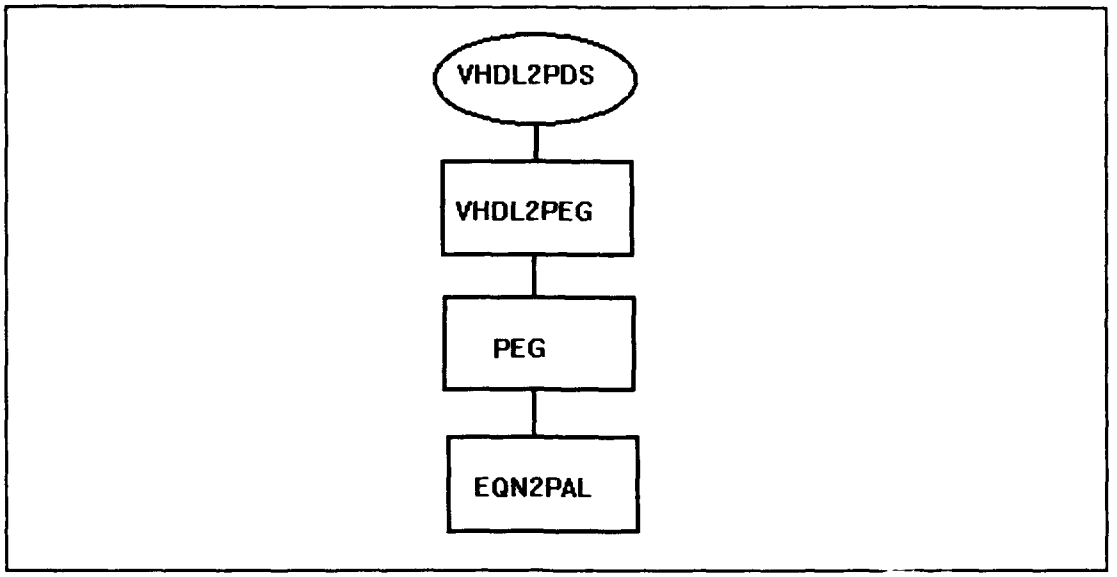

Figure 11 Programs executed by VHDL2PDS

the design, the transition equations are entered. The syntax of an **if** statement may include compound **and** statements. Variables which do not appear in the state transition "if" statements are ignored.

2. The Translation Process

When VHDL2PDS is executed from the MS-DOS command line, it takes as its argument the root name of the design. It passes this name in turn to each of the three programs it invokes: VHDL2PEG, PEG, and EQN2PDS.

a. VHDL2PEG

To the root name that it receives from VHDL2PDS, VHDL2PEG concatenates the strings ".VHD", ".PDS", ".INT", and ".PEG" which it uses to refer to the VHDL source code, the PALASM output file, and the intermediate files it creates to

```
VHDL State Machine Description
use work.translate.all;  -- Must use this library
[use work.other.all;]    -- Optionally use other
entity <entity_name> is
  port(variable1  : <type> <size>;
       ...
       CLK  : in Bit;     -- Must specify clock
       RESET: in Bit);    -- Must specify reset
end <entity_name>;
architecture dataflow_description of <entity_name> is
  begin
  <combinational_variable1> <= <expression> [after time];
  process
  type state_type is range 0 to 8; -- Need Range     --
  variable state : state_type := 0;-- Must start in 0 --
  begin
    wait on CLK until CLK'EVENT;
    if (CLK = '0') then -- Assert Outputs
      case state is
         when 0 => outputvariable1<=<expression>;
           ...
           outputvariableN<=<expression>;
         when N => outputvariable1<=<expression>;
           ...
           outputvariableN<=<expression>;
         end case;
     else -- CLK = '1' so calculate transitions
      case state is
        when 0 =>
          if (input_variable=expression) then
           state := <next_state>;
          else
            state := <other_state>;
          end if;
          ...
         when N=>
           if (input_variable=expression) then
              state := <next_state>;
            else
             state := <other_state>;
           end if
         end if;
       end case;
     end if;
    end process;
    end dataflow_description;
```

Figure 12 VHDL state machine syntax

communicate with EQN2PDS and PEG respectively. VHDL2PEG is the heart of the state machine translator. It lexically analyzes and parses the VHDL description in a similar fashion to VHDL2PAL, and it outputs information to the state machine compiler PEG. In addition, it creates the PALASM header file and provides a list of the symbols it encounters to EQN2PDS. The University of California, Berkeley's VLSI Tools reference manual [Ref 1:p. 70] documents the syntax expected by PEG. In this syntax, if a variable is to be set to a value of logic one, it is included in the output **assert** statement. If a variable is replaced by a question mark in the transition state equation, its value has no effect on the transition out of that state. VHDL2PEG maintains a table of input and output signals that are placed in the PEG file. It also translates the compound **if** statements it encounters into a corresponding PEG **case** statement by recursive calls to the parser. In addition to creating the PALASM header file with its list of pins, chip type, and combinational logic expressions, VHDL2PEG also creates an intermediate file that is read by EQN2PDS. This file contains a list of all of the signals that have been used as combinational outputs and are therefore already taken care of in the PALASM output file. EQN2PDS, which is discussed below, reads this information and uses it to avoid the problem of erroneously flagging an error when it discovers that a particular variable is never used by the state machine logic.

```
-- PEG input format
INPUTS:  var1,..., varN;

OUTPUTS: varN+1,...varN+M;

S0     :  ASSERT varN+1 varN+2;
  CASE ( var1 ... varN)
    0110 => S1;
    ???? => S0;
  ENDCASE => LOOP;
  .
  .
  .

 SL     :  ASSERT varN+2;
  CASE ( var1 ... varN)
    0101 => S2;
    ???? => S1;
  ENDCASE => LOOP;
```

Figure 13 PEG input syntax

b. PEG

PEG is a finite state machine compiler written by Gordon Hamachi at the University Of California, Berkeley under the funding of the Defense Advanced Research Projects Agency (DARPA). It is provided as part of their 1986 VLSI Tools Distribution set and is available to the public [Ref 1:p.1]. It is designed to be run under the Unix operating system and has been used on a variety of computers. During our research, we implemented PEG on an MS-DOS personal computer using the Borland Turbo C compiler. PEG takes a description of a Moore model state machine in the syntax listed in Figure 13 and it computes the state machine transition and output equations in a format known as "eqntott" that is compatible with other

Berkeley tools. More information on PEG can be found in the Berkeley VLSI Tools documentation [Ref. 1:p. 70].

c. EQN2PDS

EQN2PDS takes the "eqntott" format equations created by PEG and translates them into PALASM. It is similar in approach to VHDL2PAL and is also written in Turbo Pascal. It has a lexical analyzer and a parser along with a dictionary of tokens in both "eqntott" format and PALASM syntax. The program also reads a file created by VHDL2PEG that contains a list of signals which use purely combinational logic. This list prevents EQN2PDS flagging an error when it discovers that these declared signals are never used in the state machine transition or output equations.

V. THE XILINX LOGIC CELL ARRAY

This chapter introduces the Xilinx Logic Cell Array (LCA), a state-of-the-art field programmable gate array (FPGA). It also presents the Xilinx software tools that are used to configure a Logic Cell Array to implement a design. The information presented is derived from Ref. 1.

A. LOGIC CELL ARRAY STRUCTURE

The LCA consists of four major components: Configurable Logic Blocks (CLB's), Input/Output Blocks (IOB's), Interconnects, and Configuration Memory.

1. Configurable Logic Blocks

Configurable Logic Blocks contain the functional units that are used to implement the combinational logic portion of the user's design. The CLB's implement these boolean functions using a 32 by 1 lookup table. The CLB's are arranged in a square array with the size dependent on the chip size. Figure 14 is an example of a CLB.

2. Input Output Blocks

Input Output Blocks interface the Logic Cell Array's internal logic with its external pins. In addition, they provide registered inputs that can be used to hold the state variables in a finite state machine. These registers feature a global reset that can be used to ensure the state of all

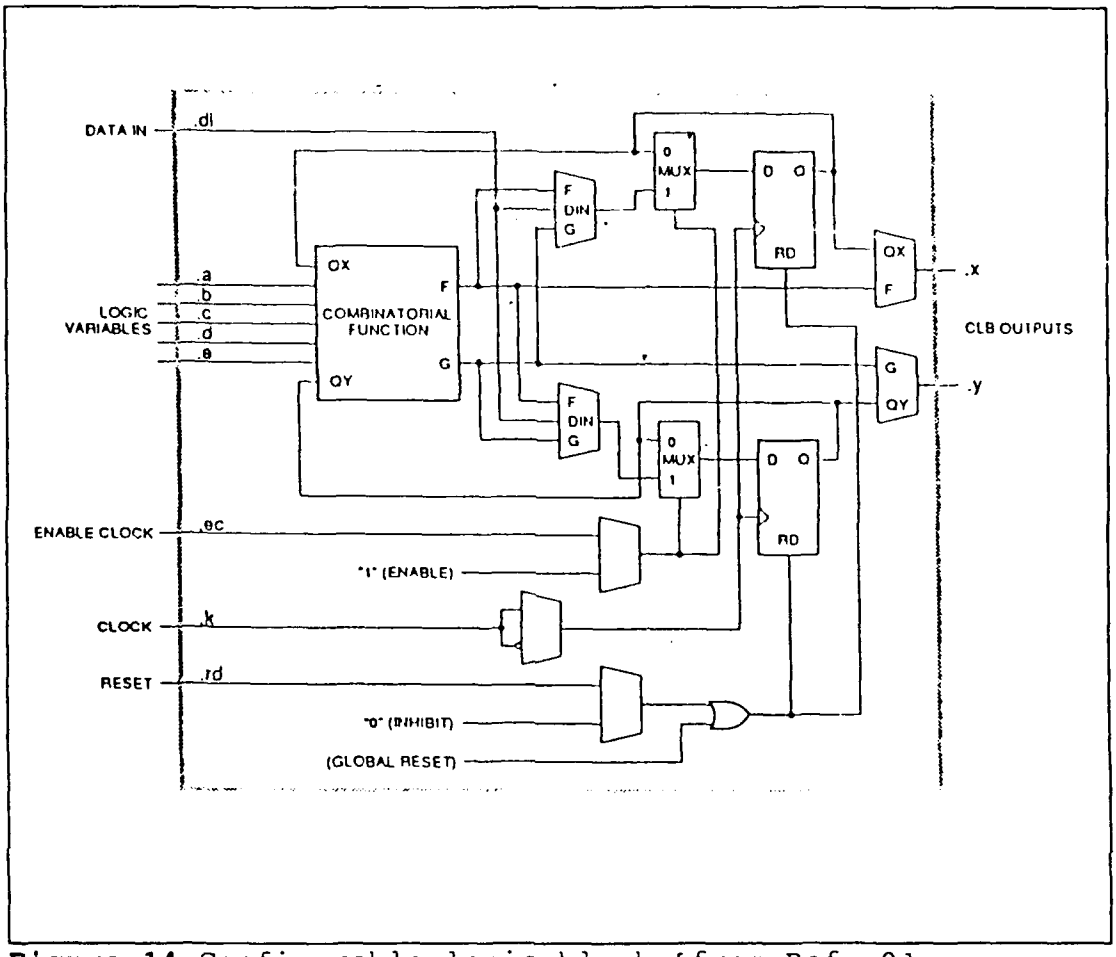

Figure 14 Configurable logic block [from Ref. 9]

flip-flops in a design as well as individual flip-flop resets. The IOB's also contain a tri-state output buffer that can be controlled by the user. Finally, the Input Output Blocks load configuration data into the LCA during the programming phase that occurs at power-up and may occur at other user-specified times. Figure 15 is an example IOB.

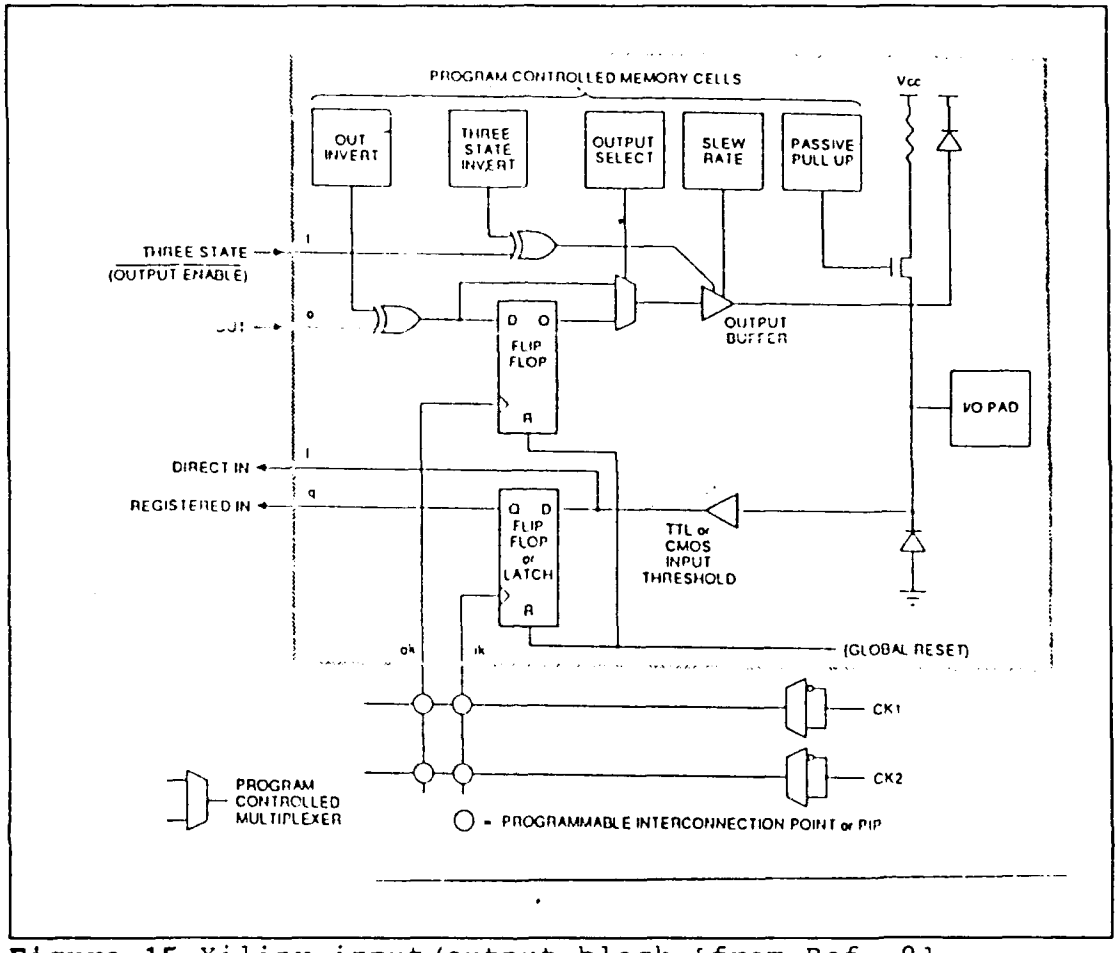

Figure 15 Xilinx input/output block [from Ref. 9]

3. Interconnects

The Logic Cell Array contains programmable interconnects that provide internal connections inside the LCA. There are three types of interconnects: general purpose, direct, and long lines. General purpose interconnects connect CLB's by means of a switching matrix. The configuration of the switching matrix is stored in configuration memory internal to the LCA. Direct interconnects connect the IOB's and CLB's that are in close proximity to each other. They are

also used to connect adjacent CLB's together. Finally, long lines interconnects bypass switching matrices and are used to transfer data over long distances across the LCA. The long lines interconnects minimize the problem of signal skew inherent in signals carried over long distances. Figure 16 shows the interconnects in a Logic Cell Array.

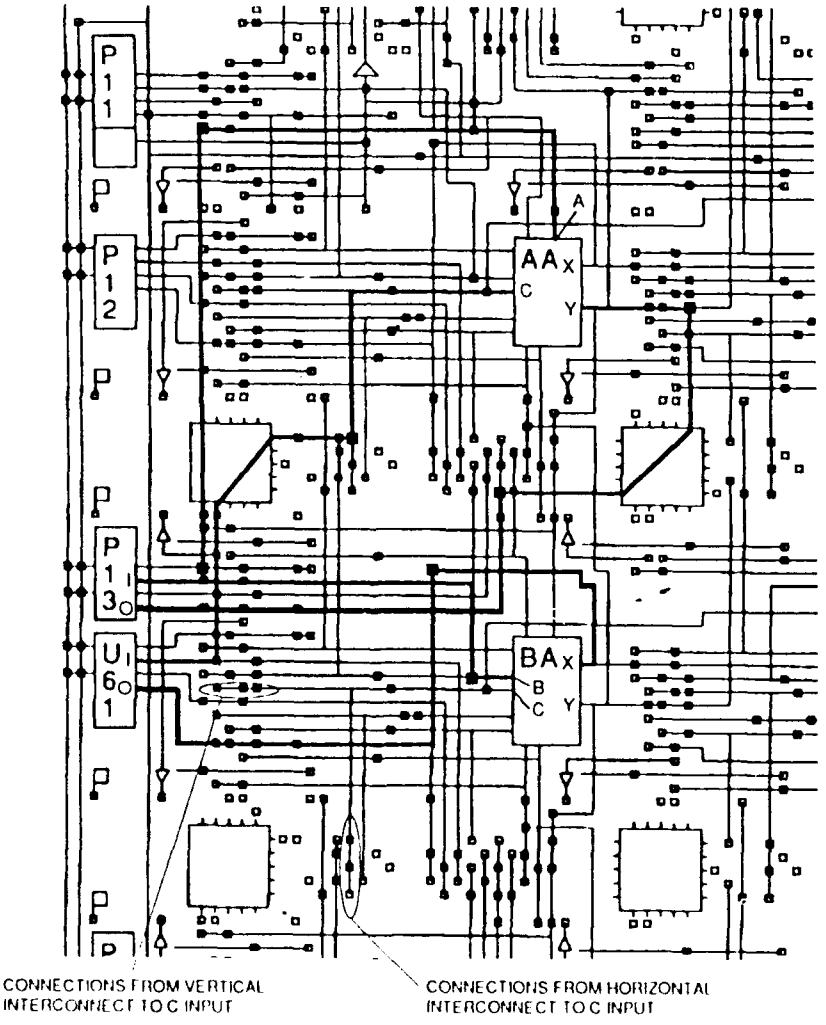

Figure 16 Interconnect and switching matrix [from Ref. 9]

4. Configuration Memory

Configuration Memory holds the user's configuration data for the CLB's, IOB's, and Interconnects. It is static memory, and is loaded from a user-supplied external data device that may be an EPROM, PROM, or other storage device. Configuration Memory may be reloaded at any time by the user, thus allowing the design to be completely changed as desired.

B. XILINX SOFTWARE TOOLS

The Xilinx development system provides software to allow the designer to configure the Logic Cell Array and prepare it for timing simulation. The software is implemented on many Unix-based workstations and there is also a version which runs on an MS-DOS personal computer. Figure 17 shows the order of these tools in the procedure pipeline used to create a design.

1. PALASM to XNF Translation

The first step in Xilinx's logic synthesis software is to translate the PALASM description to Xilinx Netlist File (XNF) format. The program PDS2XNF performs this function. The XNF file thus created may be used as a stand-alone design, or it may be merged with other XNF design files that have been created by PDS2XNF or by schematic capture. The Xilinx version of PALASM supports combinational and registered logic with optional tri-state outputs. Each registered output may use its own separate clock and its own reset line, or the

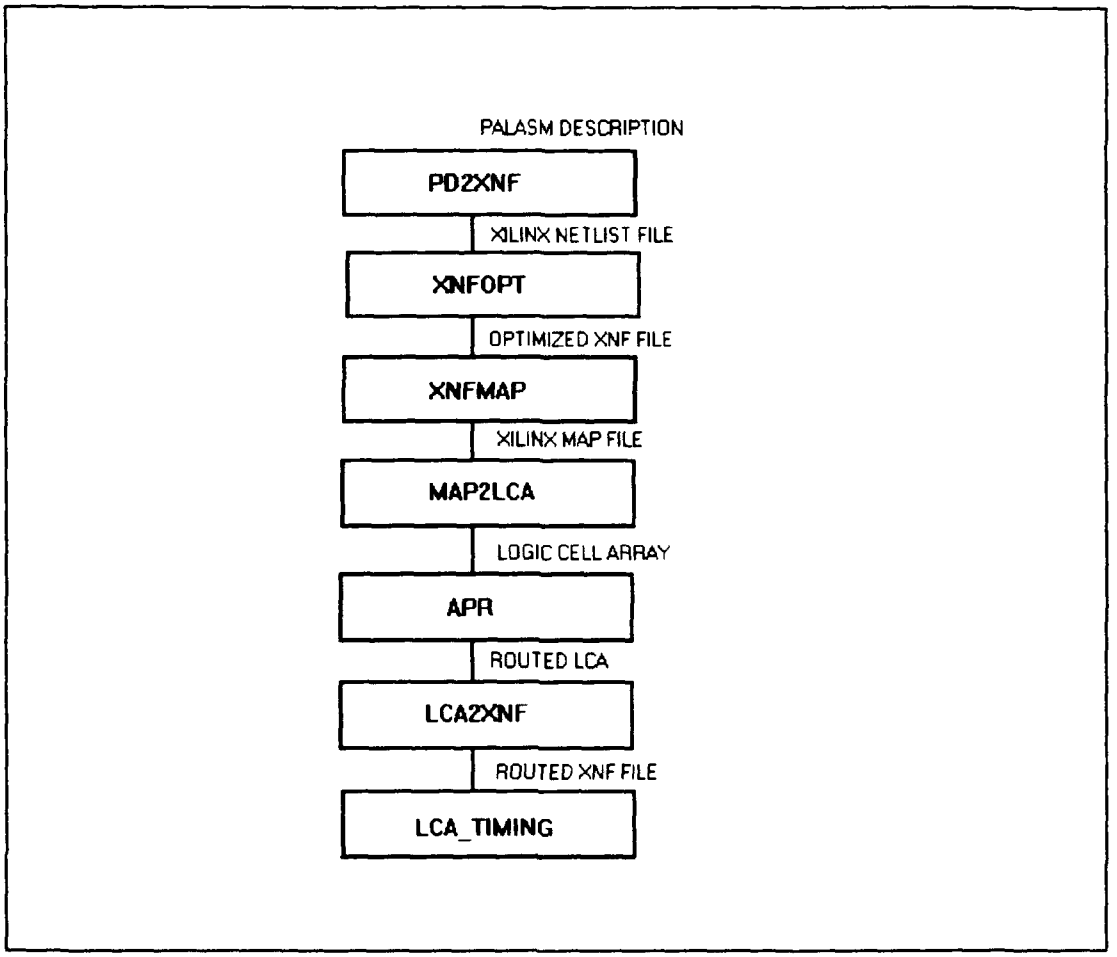

Figure 17 Xilinx logic synthesis software tools

clock and reset line may be shared with other signals. The Xilinx PALASM software supports an unlimited number of signals on the CHIP pinlist, and signal names may be up 32 characters in length. In addition, arbitrarily complex expressions are permitted in equations.

2. Optimizing the XNF File

After PDS2XNF has created an XNF file, the program XNFOPT will optimize it. This important program uses

heuristic routines to attempt to find a minimal cover under the constraint that it may not exceed the maximum fan-out of the Logic Cell Array. XNFOPT produces two XNF files: BESTCLB.XNF and BESTLVL.XNF. BESTCLB is optimized for the minimum space in the gate array and BESTLVL is optimized for speed (minimum levels). Since the optimization is heuristic, it requires significant iteration to search for the minimal cover.

3. Partitioning the Logic

After the XNF file has been optimized, XNFMAP maps the logic into the Logic Cell Array resources: CLB's and IOB's. The output of XNFMAP is a MAP file that describes these resource allocations. XNFMAP also produces a cross-reference report file (CRF) that relates the original logic elements to their mapping in the Logic Cell Array. The CRF file additionally lists errors XNFMAP encountered and it lists the optimization performed. XNFMAP performs several simple types of optimization. It removes inverter wherever possible and any other unused logic it finds. Additionally, it tries to group combinational logic functions together in such a way as to maximize the logic in each CLB and thus minimize the number of CLB's used. The program also tries to minimize the interconnections between CLB's by combining latches and flip-flops into the logic functions in the CLB's.

4. Creating the Logic Cell Array Description

The next step in the synthesis of a logic design is to create the unplaced and unrouted LCA file. The program MAP2LCA takes the MAP file created by XNFMAP and translates it into an LCA file. This LCA file describes how the design is partitioned, but it does not have information concerning which IOB's and CLB's have been chosen and how they are interconnected. MAP2LCA also produces an AKA file that gives information on signal names in the design.

5. Placement and Routing

The program APR is named for its function: automated placement and routing. This program determines the selection and interconnection of IOB's and CLB's in the Logic Cell Array through a process known as simulated annealing. It takes the LCA file created by MAP2LCA and it produces a new LCA file that contains the routing and placement information.

6. Creating the Bitstream

Once the routed LCA file is produced, the next step is to create a file that may be downloaded into an EPROM or other storage device to be used to program the Logic Cell Array. The program MAKBITS performs this function. It takes the routed LCA file and it produces the file that is then read by MAKEPROM to create the hexadecimal file required by an EPROM programmer.

7. Obtaining Timing Results

To determine the delays associated with the routed design, two programs must be run. First, LCA2XNF takes the routed LCA file and it produces an XNF file. This XNF file is then read by the batch file LCA_TIMING that creates a file named SIMSHEET that can be read by **Quicksim**, a digital hardware simulator. The information obtained from **Quicksim** may then be *back annotated* into the VHDL model that was used to create the PALASM description.

VI. THE DESIGN PROCESS

This chapter examines the process of creating digital hardware using VHDL and the synthesis tools presented earlier. It begins by introducing the Vantage Spreadsheet, a VHDL compiler/simulator. It then provides examples of combinational logic and state machine design. Figure 18 illustrates the process of digital design using VHDL. The

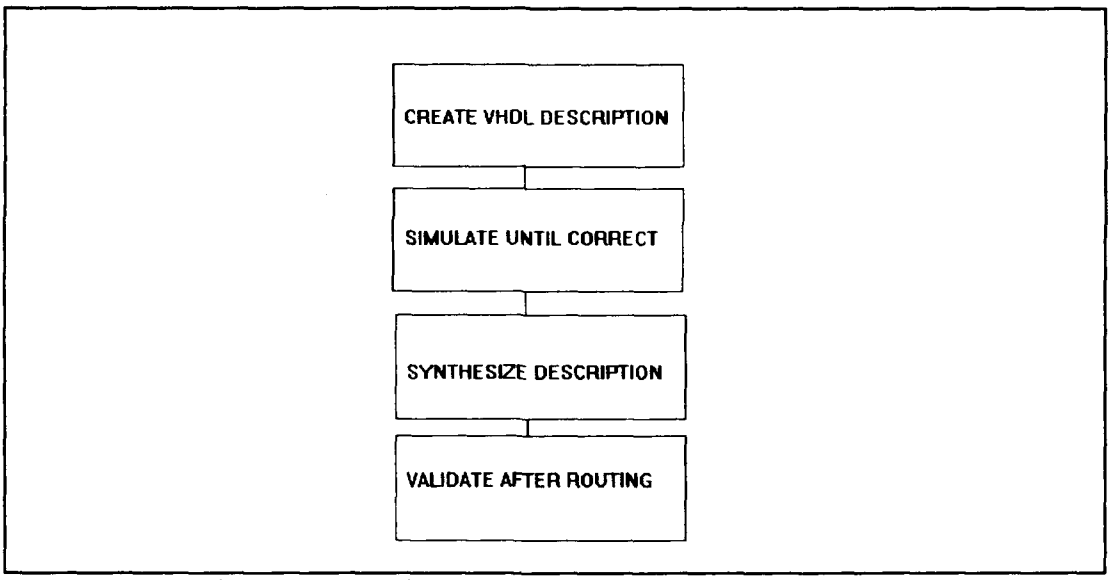

Figure 18 Digital design process using VHDL

first step is to create a description in VHDL. The design is then simulated and the source code is modified until the design is correct. The Vantage Spreadsheet can be used to perform these functions. After the design functions correctly

in VHDL, it is translated to PALASM using the programs VHDL2PAL (combinational logic) or VHDL2PDS (finite state machines) and synthesized using the Xilinx tools addressed in Chapter 5.

A. THE VANTAGE SPREADSHEET

The Vantage Spreadsheet is a CAD tool with a graphical user interface which allows the engineer to interactively create and simulate a VHDL model. Figure 19 shows the modules available in the Spreadsheet. The VHDL compiler takes VHDL source code or schematics and compiles them into libraries. The compiler obtains behavioral information about

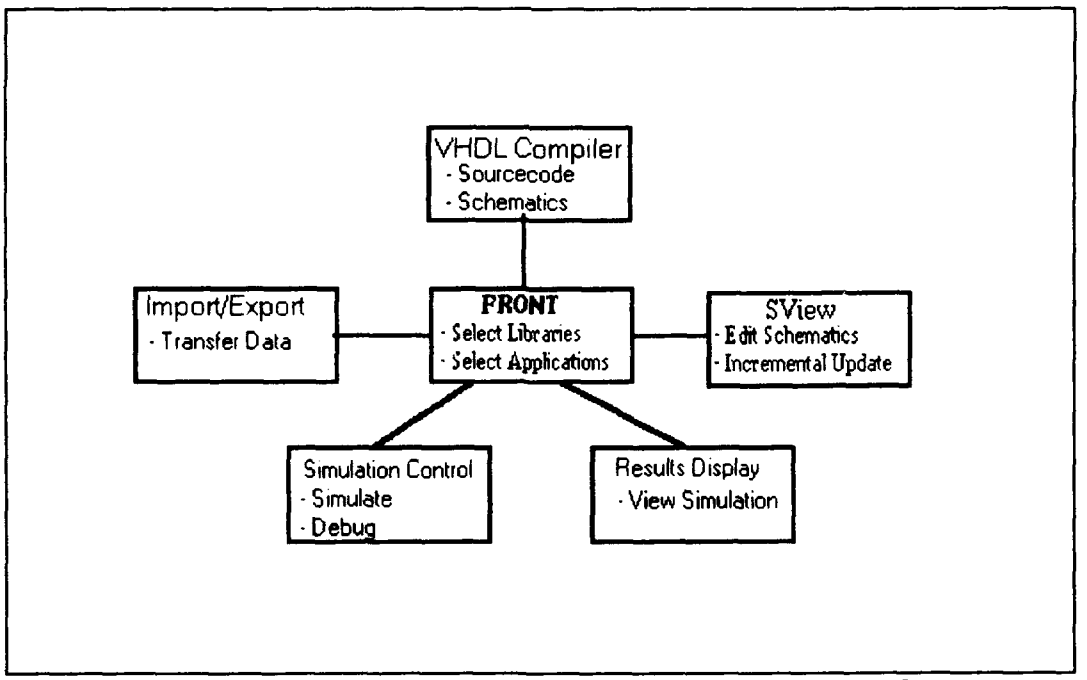

Figure 19 Vantage Spreadsheet modules [after Ref. 10]

the model from a VHDL dataflow description or behavioral description. As illustrated in Figure 20, structural information can come from a schematic or a VHDL structural description. After the compiler has successfully been run, the design may be simulated. The user can debug the design by providing stimulus to the simulation and viewing the value of variables. An example output is provided in Figure 21.

B. COMBINATIONAL LOGIC SYNTHESIS EXAMPLE

Combinational logic design is illustrated in the synthesis of and eight bit adder/subtractor. For clarity, the high order bits are omitted and the VHDL source code of the lowest order two bits in shown in Figure 22. The approach is a to create boolean equations which exhibit the desired behavior. Depending on the application, these equations may be formed from a truth table of the function to be implemented, or directly from the desired behavior. The architecture is described in the **port** statement which specifies the quantities **A** and **B** that are to be added/subtracted, an output signal **F**, a carry bit **CN**, and a mode bit **M** specifying addition or subtraction. The logic equations appear next along with an estimate of 80 nanoseconds for the delay. The equivalent PALASM equations generated by the translation software is provided in Figure 23. The PALASM code begins with a comment header. The entity name **ADDER** is specified along with the output chip type **LCA** which is used to indicate that the design

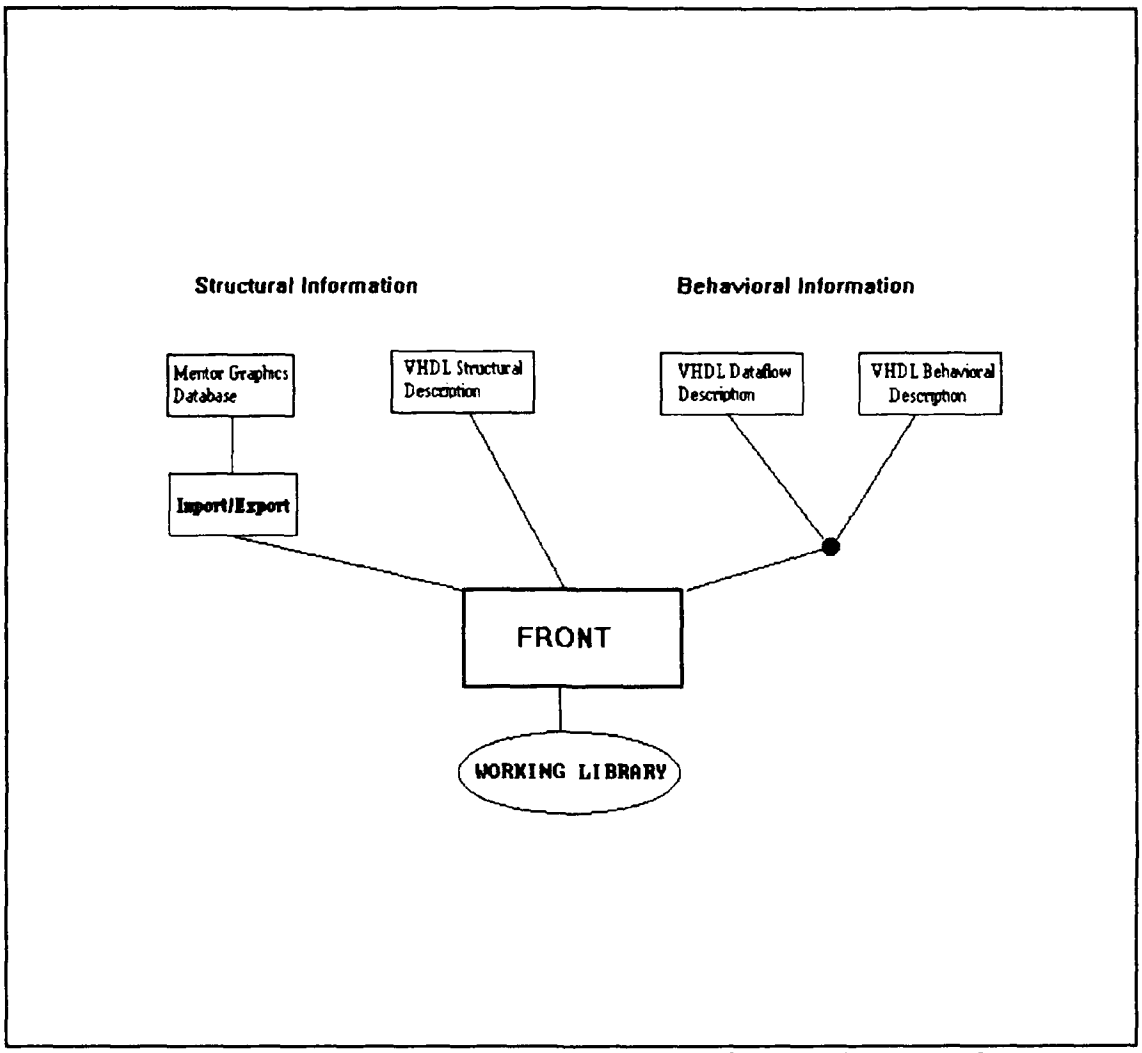

Figure 20 Vantage Spreadsheet compilation [after Ref. 10]

can be used with any members of the Logic Cell Array family. The pin declarations follow, with the signals of type *byte* having pins declared for each of the bits in the signal running from zero through seven. The PALASM logic equations follow, each having being translated to the correct syntax.

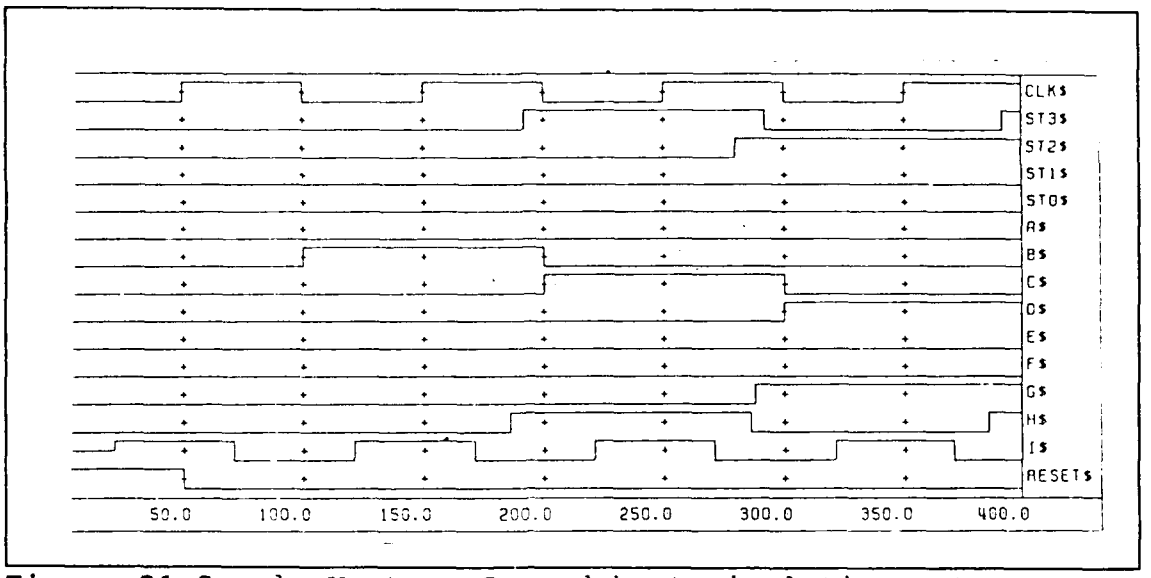

Figure 21 Sample Vantage Spreadsheet simulation output

C. STATE MACHINE SYNTHESIS EXAMPLE

The state diagram for a simple state machine is provided in Figure 24. A VHDL representation of this state machine is presented in three pieces. The first piece is displayed in Figure 25. It consists of a comment header and an **entity** description which specifies the signal names and whether each is an input to the state machine or an output from it. The reserved signals **CLK** and **RESET** are also declared. The next piece of the VHDL description is presented in Figure 26. It begins with the combinational logic associated with the state machine. One equation is illustrated, but any number of equations may be provided as desired by the user. The state type declaration specifies that there are nine states available in the description (zero through eight). The

```
-------------------------------------------------------
-- "adder.vhd"                                       --
-- This VHDL program simulates an adder subtractor.  --
-- It is written in a subset of VHDL which can be    --
-- synthesized into a functional gate array using    --
-- VHDL2PAL which translates it into the subset of   --
-- PALASM recognized by the Xilinx LCA gate array    --
-- system.                                           --
-- This examples includes only the lowest two bits   --
-------------------------------------------------------
use work.translate.all;
entity ADDER is
  port(M    : in Bit;
       A    : in Byte;
       B    : in Byte;
       CN   : in Bit;
       F    : out Byte);
end ADDER;

architecture dataflow_description of adder is
begin
  process
  begin

F(0) <= transport (not CN) xor ((not(A(0) or (((not M) and
B(0)) or (M and (not B(0)))))) xor (not (A(0) and (((not M)
and B(0))or (M and (not B(0)))))) after 80NS;

F(1) <= transport (not ((not (A(0) or (((not M) and B(0))
or (M and (not B(0)))))) or ((not (A(0)  and (((not M) and
B(0)) or (M and (not B(0)))))) and CN))) xor ((not (A(1) or
((( not M) and B(1)) or (M and ( not B(1)))))) xor (not
(A(1) and (((not M) and B(1)) or (M and (not B(1)))))) )
after 80NS;

  end process;
end dataflow_description;
```

Figure 22 VHDL description of Adder/Subtractor

initial state may be any number; the translator computes the difference between starting and ending numbers to determine the number of state variables to allocate. The output equations are the next major section in the description. Each

```
;-------------------------------------------------------
TITLE        ADDER.PDS
PATTERN      A
REVISION     REV 1.0
AUTHOR       Automated
COMPANY      U.S. Naval Postgraduate School
DATE         8/31/1991
;
CHIP ADDER LCA
;-------------------------------------------------------
;               PIN Declarations
;
M A0 A1 A2 A3 A4 A5 A6 A7 B0 B1 B2 B3 B4 B5 B6 B7 CN F0
F1
F2 F3 F4 F5 F6 F7 ;
;-------------------------------------------------------
;
EQUATIONS
F0=(/CN):+:(((/(A0+(((/M)*B0)+(M*(/B0))))):+:(/(A0*(((/M)*
B0)+(M*(/B0)))
)))

F1=(/(((/(A0+(((/M)*B0)+(M*(/B0))))))+(((/(A0*(((/M)*B0)+(M*
(/B0)))))*CN))
):+:(((/(A1+(((/M)*B1)+(M*(/B1))))):+:(/(A1*(((/M)*B1)+(M*
(/B1))))))
```

Figure 23 PALASM translation of VHDL combinational circuit

state must specify the output of all variables. If the output value of a specific variable is unimportant in a particular state, its output must be specified arbitrarily. Figure 27 contains the last portion of the VHDL description: the transition equations. As seen in this figure, only signals which are required for a state transition need to be specified. When multiple signals are required for a transition, they may be specified through the **AND** conjunction

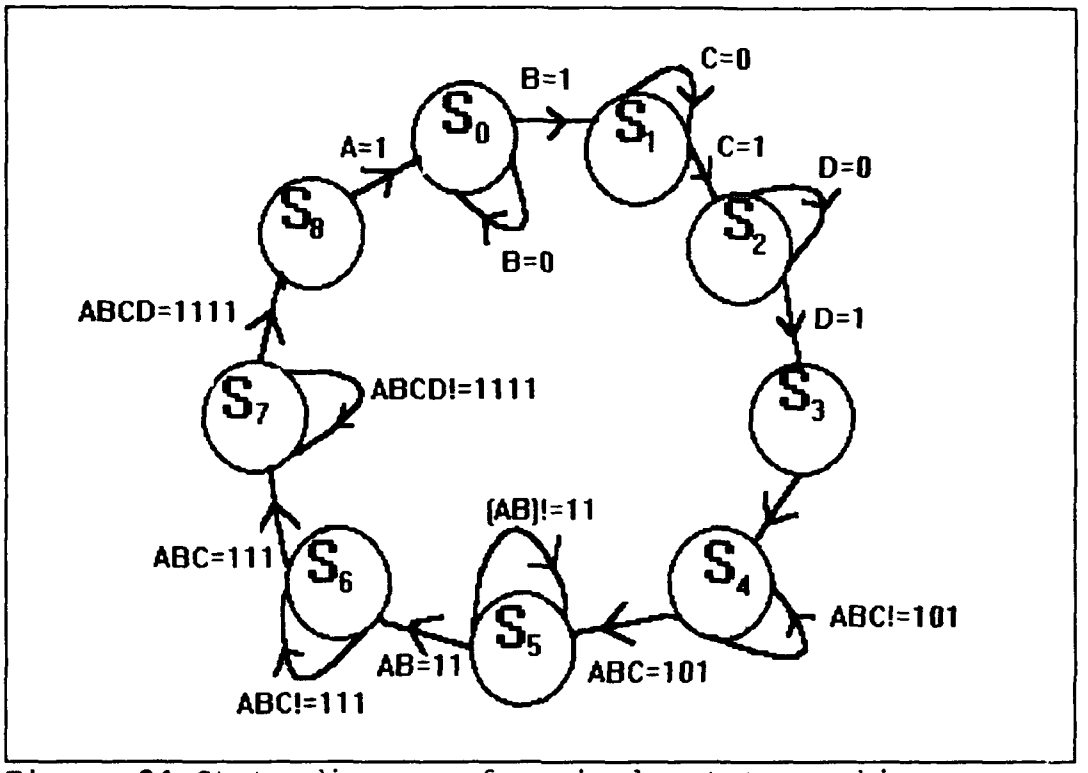

Figure 24 State diagram of a simple state machine

in the **if** statement. This is illustrated in the figure in the transition equations for state seven. When the state machine proceeds to the next state regardless of the input, no **if** statement is included as seen in the transition out of state three. Finally, an **else** statement is used to specify the next state if the conditions in the **if** statement are not met. This example clearly demonstrates the ease with which state machine descriptions may be created using the synthesis tools. Any degree of complexity may be specified, up to the limitations imposed by the size of the gate array. Finally, the PALASM representation of the state machine created by the translation

```
-- ------------------------------------------
--                                          --
-- "Simple State Machine"  This state machine--
-- is written in VHDL.  It simulates the    --
-- behavior of a simple state machine. It is --
-- written in the subset of VHDL which can be--
-- translated in PALASM for synthesis.      --
-- It uses standard logic and the translate  --
-- libraries and includes the signals CLK    --
-- and RESET which are required when using   --
-- registered logic.                        --
------------------------------------------
use std.std_logic.ALL;
use work.translate.all;
entity STATE_MACHINE is
  port(A     : in Bit;
       B     : in Bit;
       C     : in Bit;
       D     : in Bit;
       E     : out Bit;
       F     : out Bit;
       G     : out Bit;
       H     : out Bit;
       I     : out Bit;
       CLK   : in Bit;
       RESET: in Bit);
end STATE_MACHINE;
```

Figure 25 VHDL state machine entity

programs is presented in Figure 28. It is markedly different in appearance from its VHDL counterpart. Like the PALASM combinational example above, the PALASM state machine description begins with a comment header. The **CHIP** statement then specifies the design name obtained from the VHDL description and that a Logic Cell Array is the target device. Pins are declared corresponding to the signals in the VHDL description. The special signals **CLK** and **RESET** appear here

```
architecture dataflow_description of STATE_MACHINE is
  begin
  -- Combinational assignments
  I <= not CLK after 40 NS;
  process
  type state_type is range 0 to 8;
  variable state : state_type := 0; -- Start S0
  begin
    wait on CLK until CLK'EVENT;
    if (CLK = '0') then -- Assert Outputs
      case state is
         when 0 => E<='0';
                   F<='0';
                   G<='0';
                   H<='0';
         when 1 => E<='0';
                   F<='0';
                   G<='0';
                   H<='1';
         when 2 => E<='0';
                   F<='0';
                   G<='1';
                   H<='0';
         when 3 => E<='0';
                   F<='0';
                   G<='1';
                   H<='1';
         when 4 => E<='0';
                   F<='1';
                   G<='0';
                   H<='0';
         when 5 => E<='0';
                   F<='1';
                   G<='0';
                   H<='1';
         when 6 => E<='0';
                   F<='1';
                   G<='1';
                   H<='0';
         when 7 => E<='0';
                   F<='1';
                   G<='1';
                   H<='1';
         when 8 => E<='1';
                   F<='0';
                   G<='0';
                   H<='0';
      end case;
```

Figure 26 VHDL state machine output equations

along with four state variables, sufficient for the nine states used in the VHDL description. The output equations follow. Since the finite state machine is a Moore machine, these are solely a function of the state variables. Finally, each pin allocated for a registered state variable has a corresponding reset and clock line. The translation software allocates the same reset and clock for every state variable in the state machine. While the PALASM code is significantly shorter, it is functionally equivalent to the VHDL description and produces the same output when simulated.

```
else -- CLK = '1' so calculate transitions
 case state is
   when 0 =>if (B='1') then state := 1;
      else state := 0;
      end if;
    when 1=>if (C='1') then state := 2;
      else state:=1;
      end if;
    when 2=>if (D='1') then state := 3;
      else state := 2;
      end if;
    when 3=>state :=0;
    when 4=>if (A='1') and (B='0') and (C='1') then
         state := 5;
      else state := 4;
      end if;
    when 5=>if (A='1') and (B='1') then
         state := 6;
      else state := 5;
      end if;
    when 6=>if (A='1') and (B='1') and (C='1') then
         state := 7;
      else state := 6;
      end if;
     when 7=>
     if (A='1') and (B='1') and (C='1') and (D='1') then
         state := 8;
      else
          state := 7;
      end if;
     when 8=>if (A='1') then state := 1;
      else state := 8;
      end if;
   end case;
 end if;
end process:
end dataflow_description;
```

Figure 27 VHDL state machine transition equations

```
;-----------------------------------------------
TITLE        STATE_MACHINE.PDS
PATTERN      A
REVISION     REV 1.0
AUTHOR       Automated
COMPANY      U.S. Naval Postgraduate School
DATE         7/31/1991
CHIP STATE_MACHINE LCA
;-----------------------------------------------
;                  PIN Declarations
A B C D E F G H I CLK RESET ST0 ST1 ST2 ST3 ;
;-----------------------------------------------
EQUATIONS
;----- Combinational Equations ---------------
I=/CLK
;- State Equations ---------------------------
ST3:=(A*ST0*/ST1*/ST2*/ST3)+(/A*B*C*D*/ST0*ST1*ST2*ST3)+(/
B*C*D*/ST0*ST1*ST2*ST3)+(/C*D*/ST0*ST1*ST2*ST3)+(/D*/ST0*S
T1*ST2*ST3)+(A*B*C*/ST0*ST1*ST2*/ST3)+(/A*B*/ST0*ST1*/ST2*
ST3)+(/B*/ST0*ST1*/ST2*ST3)+(A*/B*C*/ST0*ST1*/ST2*/ST3)+(D
*/ST0*/ST1*ST2*/ST3)+(/C*/ST0*/ST1*/ST2*ST3)+(B*/ST0*/ST1*
/ST2*/ST3)

ST2:=(/A*B*C*D*/ST0*ST1*ST2*ST3)+(/B*C*D*/ST0*ST1*ST2*ST3)
+(/C*D*/ST0*ST1*ST2*ST3)+(/D*/ST0*ST1*ST2*ST3)+(/ST0*ST2*/
ST3)+(A*B*/ST0*ST1*/ST2*ST3)+(C*/ST0*/ST1*/ST2*ST3)

ST1:=(/A*B*C*D*/ST0*ST1*ST2*ST3)+(/B*C*D*/ST0*ST1*ST2*ST3)
+(/C*D*/ST0*ST1*ST2*ST3)+(/D*/ST0*ST1*ST2*ST3)+(/ST0*ST1*/
ST3)+(/ST0*ST1*/ST2*ST3)ST0:=(/A*ST0*/ST1*/ST2*/ST3)+(A*B*
C*D*/ST0*ST1*ST2*ST3)

E=(ST0*/ST1*/ST2*/ST3)
F=(/ST0*ST1)
G=(/ST0*ST2)
H=(/ST0*ST3)
ST0.RSTF = RESET
ST0.CLKF = CLK
ST1.RSTF = RESET
ST1.CLKF = CLK
ST2.RSTF = RESET
ST2.CLKF = CLK
ST3.RSTF = RESET
ST3.CLKF = CLK
```

Figure 28 State machine translated to PALASM

VII. CONCLUSIONS AND AREAS FOR FUTURE RESEARCH

This thesis examined top down digital design using VHDL. It defined a subset of VHDL which can be used to model combinational logic and finite state machines and it presented software tools that synthesize this subset into a gate array. The thesis reviewed the hardware synthesis language PALASM, and examined the process of translating VHDL to PALASM. It explored the Xilinx Logic Cell Array, and it examined the software used to configure this gate array. Additionally, the thesis presented the Vantage Spreadsheet VHDL compiler/simulator, and introduced its use in facilitating top-down digital design. It is very clear that automatic synthesis greatly enhances the usefulness of VHDL in digital hardware design. While VHDL is in itself an excellent tool for creating digital hardware, the language is significantly more valuable when coupled with synthesis tools. The software created in our research can be built upon. More features can be implemented in the VHDL subset accepted by the tools. The automation of the process of back-annotating timing parameters may be explored. Moreover, the tools created in this research may be used to create new designs. These designs may be optimized by the Xilinx system for space or speed and the resulting hardware may be compared against functionally

similar designs created by schematic capture. Finally, a topic of significant future research is the synthesis of systems that do not have an implied architecture. While combinational logic designs and finite state machines are of great practical value, true behavioral synthesis will fundamentally alter the way engineers envision the task of digital hardware design. It will provide a greater level of abstraction which will mean shorter development time, greater portability of designs across different technologies, and lower costs than ever before. Finally, automatic digital hardware synthesis using hardware description languages is a design paradigm which is gaining momentum in industry. It offers better designs in less time and with less cost and is therefore the method of choice for digital hardware design.

LIST OF REFERENCES

1. Walter S. Scott, Robert N. Mayo, Gordon Hamachi, John K. Ousterhout, eds., *1986 VLSI Tools: Still More Works by the Original Artists*, University of California, Berkeley, 1985.
2. Franklin P. Prosser and David E. Winkel, *The Art of Digital Design*, Prentice-Hall, Englewood Cliffs, NJ, 1987.
3. Mario R. Barbacci and Takao Uehara, "Computer Hardware Description Languages: The Bridge Between Softwar- and Hardware," *IEEE Computer*, Vol. 18, No. 2, 1985.
4. Moe Shahdad, Roger Lipsett, Erich Marschner, Kellye Sheehan, Howard Cohen, Ron Waxman, and Dave Ackley, "VHSIC Hardware Description Language," *IEEE Computer*, Vol. 18, No. 2, 1985.
5. Roger Lipsett, Carl Schaefer, and Cary Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, Norwell, MA, 1990.
6. Steve Carlson, *Introduction to HDL-Based Design Using VHDL*, Synopsys, Inc., Mountain View, CA 1990.
7. Wolfgang Glunz and Gabi Umbreit, "VHDL for High-Level Synthesis of Digital Systems," *Proc. of First European Conf. on VHDL,* Institut Mediterraneen de Technologie, Marseille, France, Volume I, 1990.

8. Alfred V. Aho and Jeffrey D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, MA, 1979.

9. Xilinx Inc., *The Programmable Gate Array User's Guide*, Xilinx Inc., San Jose, CA 1989.

10. Vantage Analysis Systems, Inc., *The Vantage Spreadsheet User's Guide*, Vantage Analysis Systems, Inc., Fremont, CA 1990.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Professor C. H. Lee, Code EC/Le Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	2
5.	Professor C. Yang, Code EC/Ya Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
6.	LT J.W. Ailes, U.S. Navy Surface Warfare Officer's School NETC, Newport, RI 02840	2